MutexVariables and Deadlocks

When multiple threads work with different data structures each of which is protected by a separate mutex variable, caution has to be taken to avoid deadlocks. A deadlock may occur if the threads use a different order for locking the mutex variables. This can be seen for two threads T_1 and T_2 and two mutex variables *ma* and *mb* as follows:
thread T_1 first locks *ma* and then *mb*;
thread T_2 first locks *mb* and then *ma*.
If T_1 is interrupted by the scheduler of the runtime system after locking *ma* such that T_2 is able to successfully lock *mb*, a deadlock occurs:
T_2 will be blocked when it is trying to lock *ma*, since *ma* is already locked by T_1; similarly, T_1 will be blocked when it is trying to lock *mb* after it has been woken up again, since *mb* has already been locked by T_2. In effect, both threads are blocked forever and are mutually waiting for each other. The occurrence of deadlocks can be avoided by using a ***fixed locking order*** for all threads or by employing a ***backoff strategy***.

Fixed Locking Order

When using a **fixed locking order**, each thread locks the critical mutex variables always in the same predefined order. Using this approach for the example above, thread T_2 must lock the two mutex variables *ma* and *mb* in the same order as T_1, e.g., both threads must first lock *ma* and then *mb*. The deadlock described above cannot occur now, since T_2 cannot lock *mb* if *ma* has previously been locked by T_1. To lock *mb*, T_2 must first lock *ma*. If *ma* has already been locked by T_1, T_2 will be blocked when trying to lock *ma* and, hence, cannot lock *mb*. The specific locking order used can in principle be arbitrarily selected, but to avoid deadlocks it is important that the order selected is used throughout the entire program. If this does not conform to the program structure, a backoff strategy should be used.

Backoff Strategy

When using a **backoff strategy**, each participating thread can lock the mutex variables in its individual order, and it is not necessary to use the same predefined order for each thread. But a thread must back off when its attempt to lock a mutex variable fails. In this case, the thread must release all mutex variables that it has previously locked successfully. After the backoff, the thread starts the entire lock procedure from the beginning by trying to lock the first mutex variable again. To implement a backoff strategy, each thread uses pthread mutex lock() to lock its first mutex variable and pthread mutex trylock() to lock the remaining mutex variables needed. If pthread mutex trylock() returns EBUSY, this means that this mutex variable is already locked by another thread. In this case, the calling thread releases all mutex variables that it has previously locked successfully using pthread mutex unlock().

Condition Variables

Mutex variables are typically used to ensure mutual exclusion when accessing global data structures concurrently. But mutex variables can also be used to wait for the occurrence of a specific condition which depends on the state of a global data structure and which has to be fulfilled before a certain operation can be applied. An example might be a shared buffer from which a consumer thread can remove entries only if the buffer is not empty. To apply this mechanism, the shared data structure is protected by one or several mutex variables, depending on the specific situation. To check whether the condition is fulfilled, the executing thread locks the mutex variable(s) and then evaluates the condition. If the condition is fulfilled, the intended operation can be performed. Otherwise, the mutex variable(s) are released again and the thread repeats this procedure again at a later time. This method has the drawback that the thread which is waiting for the condition to be fulfilled may have to repeat the evaluation of the condition quite often before the condition becomes true. This consumes execution time (*active waiting*), in particular because the mutex variable(s) have to be locked before the condition can be evaluated. To enable a more efficient method for waiting for a condition, Pthreads provide condition variables.

A **condition variable** is an opaque data structure which enables a thread to wait for the occurrence of an arbitrary condition without active waiting. Instead, a sig-naling mechanism is provided which blocks the executing thread during the waiting time, so that it does not consume CPU time. The waiting thread is woken up again as soon as the condition is fulfilled. To use this mechanism, the executing thread must define a condition variable and a mutex variable. The mutex variable is used to protect the evaluation of the specific condition which is waiting to be fulfilled. The use of the mutex variable is necessary, since the evaluation of a condition usually requires to access shared data which may be modified by other threads concurrently.
A condition variable has type pthread_cond_t. After the declaration or the dynamic generation of a condition variable, it must be initialized before it can be used. This can be done dynamically by calling the function:
int pthread_cond_init (pthread _cond _t *cond, const pthread_condattr_t *attr)

where cond is the address of the condition variable to be initialized and attr is the address of an attribute data structure for condition variables. Using attr=NULL leads to an initialization with the default attributes. For a condition variable cond that has been declared statically, the initialization can also be obtained by using the PTHREAD_COND_INITIALIZER initialization macro. This can also be done directly with the declaration.

Pthread_cond_t cond = PTHREAD_COND_INITIALIZER.

The initialization macro cannot be used for condition variables that have been gener-ated dynamically using, e.g., malloc(). A condition variable cond that has been initialized with pthread_cond_init() can be destroyed by calling the function

int pthread_cond_destroy (pthread_cond_t *cond)

if it is no longer needed. In this case, the runtime system can free the information stored for this condition variable. Condition variables that have been initialized stat-ically with the initialization macro do not need to be destroyed.

Each condition variable must be uniquely associated with a specific mutex vari-able. All threads which wait for a condition variable at the same time must use the same associated mutex variable. It is not allowed that different threads asso-ciate different mutex variables with a condition variable at the same time. But a mutex variable can be associated with different condition variables. A condition variable should only be used for a single condition to avoid deadlocks or race con-ditions . A thread must first lock the associated mutex variable mutex with pthread_mutex_lock() before it can wait for a specific condition to be fulfilled using the function.

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)

where cond is the condition variable used and mutex is the associated mutex vari-able. The condition is typically embedded into a surrounding control statement. A standard usage pattern is

```
pthread_mutex_lock (&mutex);
while (!condition())
pthread_cond_wait (&cond, &mutex);
compute something();
pthread_mutex_unlock (&mutex);
```

The evaluation of the condition and the call of pthread cond wait() are pro-tected by the mutex variable mutex to ensure that the condition does not change between the evaluation and the call of pthread_cond_wait(), e.g., because another thread changes the value of a variable that is used within the condition. Therefore, each thread must use this mutex variable mutex to protect the manip-ulation of each variable that is used within the condition. Two cases can occur for this usage pattern for condition variables:

If the specified condition is fulfilled when executing the code segment from above, the function pthread_cond_wait() is **not** called. The executing thread releases the mutex variable and proceeds with the execution of the suc-ceeding program part.

If the specified condition is not fulfilled, pthread_cond_wait() is called. This call has the effect that the specified mutex variable mutex is implicitly released and that the executing thread is blocked, waiting for the condition vari-able until another thread sends a

signal using pthread_cond_signal() to notify the blocked thread that the condition may now be fulfilled. When the blocked thread is woken up again in this way, it implicitly tries to lock the mutex variable mutex again. If this is owned by another thread, the woken-up thread is blocked again, now waiting for the mutex variable to be released. As soon as the thread becomes the owner of the mutex variable mutex, it continues the execution of the program. In the context of the usage pattern from above, this results in a new evaluation of the condition because of the while loop.

In a Pthreads program, it should be ensured that a thread which is waiting for a condition variable is woken up only if the specified condition is fulfilled. Nev-ertheless, it is useful to evaluate the condition again after the wake up because there are other threads working concurrently. One of these threads might become the owner of the mutex variable before the woken-up thread. Thus the woken-up thread is blocked again. During the blocking time, the owner of the mutex variable may modify common data such that the condition is no longer fulfilled. Thus, from the perspective of the executing thread, the state of the condition may change in the time interval between being woken up and becoming owner of the associated mutex variable. Therefore, the thread must again evaluate the condition to be sure that it is still fulfilled. If the condition is fulfilled, it cannot change before the executing thread calls pthread_mutex_unlock() or pthread_cond_wait() for the same condition variable, since each thread must be the owner of the associated mutex variable to modify a variable used in the evaluation of the condition.
Pthreads provide two functions to wake up (*signal*) a thread waiting on a condi-tion variable:

int pthread_cond_signal (pthread_cond_t *cond)
int pthread_cond_broadcast (pthread_cond_t *cond).

A call of pthread_cond_signal() wakes up a *single* thread waiting on the condition variable cond. A call of this function has no effect, if there are no threads waiting for cond. If there are several threads waiting for cond, one of them is selected to be woken up. For the selection, the priorities of the wait-ing threads and the scheduling method used are taken into account. A call of pthread_cond_broadcast() wakes up *all* threads waiting on the condition variable cond. If several threads are woken up, only one of

them can become owner of the associated mutex variable. All other threads that have been woken up are blocked on the mutex variable.
The functions pthread_cond_signal() and pthread_cond_ broadcast() should only be called if the condition associated with cond is fulfilled. Thus, before calling one of these functions, a thread should evaluate the condition. To do so safely, it must first lock the mutex variable associated with the condition variable to ensure a consistent evaluation of the condition. The actual call of pthread_cond_signal() or pthread_cond_broadcast() does not need to be protected by the mutex variable. Issuing a call without protection by the mutex variable has the drawback that another thread may become the owner of the mutex variable when it has been released after the evaluation of the condi-tion, but before the signaling call. In this situation, the new owner thread can modify shared variables such that the condition is no longer fulfilled. This does not lead to an error, since the woken-up thread will again evaluate the condi-tion.
The advantage of not protecting the call of pthread_cond_ signal() or pthread_cond_broadcast() by the mutex variable is the chance that the mutex variable may not have an owner when the waiting thread is woken up. Thus, there is a chance that this thread becomes the owner of the mutex variable without waiting. If mutex protection is used, the signaling thread is the owner of the mutex variable when the signal arrives, so the woken-up thread must block on the mutex variable immediately after being woken up.
To wait for a condition, Pthreads also provide the function

int pthread_cond_timedwait(pthread_cond_t *cond,
pthread_mutex_t *mutex,
const struct timespec *time)

The difference from pthread_cond_wait() is that the blocking on the condi-tion variable cond is ended with return value ETIMEDOUT after the specified time interval time has elapsed.

Java Threads

Java supports the development of multi-threaded programs at the language level. Java provides language constructs for the synchronized execution of program parts and supports the creation and management of threads by predefined classes.

Thread Generation in Java

Each Java program in execution consists of at least one thread of execution, the *main thread*. This is the thread which executes the main() method of the class which has been given to the Java Virtual Machine (JVM) as start argument. More user threads can be created explicitly by the main thread or other user threads that have been started earlier. The creation of threads is supported by the predefined class Thread from the standard package java.lang. This class is used for the representation of threads and provides methods for the creation and management of threads. The interface Runnable from java.lang is used to represent the program code executed by a thread; this code is provided by a run() method and is executed asynchronously by a separate thread. There are two possibilities to arrange this: inheriting from the Thread class or using the interface Runnable.

Inheriting from the Thread Class

One possibility to obtain a new thread is to define a new class NewClass which inherits from the predefined class Thread and which defines a method run() containing the statements to be executed by the new thread. The run() method defined in NewClass overwrites the predefined run() method from Thread. The Thread class also contains a method start() which creates a new thread executing the given run() method. The newly created thread is executed asynchronously with the generating thread. After the execution of start() and the creation of the new thread, the control will be immediately returned to the generating thread. Thus, the generating thread resumes execution usually before the new thread has terminated, i.e., the generating thread and the new thread are executed concurrently with each other. The new thread is terminated when the execution of the run() method has been finished

```
import java.lang.Thread;
public class NewClass extends Thread { // inheritance
  public void run() {
  // overwriting method run() of class Thread
    System.out.println("hello from new thread");
  }
  public static void main (String args[]) {
    NewClass nc = new NewClass();
    nc.start();
  }
}
```

Fig 1: Thread creation by overwriting the run() method of the Thread class

This mechanism for thread creation is illustrated in the above Fig 1 with a class NewClass whose main() method generates an object of NewClass and whose run() method is activated by calling the start() method of the newly created object. Thus, thread creation can be performed in two steps:

definition of a class NewClass which inherits from Thread and which defines a run() method for the new thread; instantiation of an object nc of class NewClass and activation of nc. start().

The creation method just described requires that the class NewClass inherits from Thread. Since Java does not support multiple inheritance, this method has the drawback that NewClass cannot be embedded into another inheritance hierarchy. Java provides interfaces to obtain a similar mechanism as multiple inheritance. For thread creation, the interface Runnable is used.

Using the Interface Runnable

The interface Runnable defines an abstract run() method as follows:

```
public interface Runnable
{
public abstract void run();
}
```

The predefined class Thread implements the interface Runnable. Therefore, each class which inherits from Thread, also implements the interface Runnable. Hence, instead of inheriting from Thread the newly defined class NewClass can directly implement the interface Runnable.
This way, objects of class NewClass are not thread objects. The creation of a new thread requires the generation of a new Thread object to which the object NewClass is passed as parameter. This is obtained by using the constructor

public Thread (Runnable target).

Using this constructor, the start() method of Thread activates the run() method of the Runnable object which has been passed as argument to the con-structor. This is obtained by the run() method of Thread which is specified as fol-lows:

```
public void run()
{
if (target != null)
target.run();
}
```

After activating start(), the run() method is executed by a separate thread which runs asynchronously with the calling thread. Thus, thread creation can be performed by the following steps:
definition of a class NewClass which implements Runnable and which defines a run() method containing the code to be executed by the new thread; instantiation of a Thread object using the constructor Thread (Runnable target) and of an object of NewClass which is passed to the Thread constructor; activation of the start() method of the Thread object.

```
import java.lang.Thread;
public class NewClass implements Runnable {
  public void run() {
    System.out.println("hello from new thread");
  }
  public static void main (String args[]) {
    NewClass nc = new NewClass();
    Thread th = new Thread(nc);
    th.start(); // start() activates nc.run() in a new thread
  }
}
```

Fig 2: Thread creation by using the interface `Runnable` based on the definition of a new class `NewClass`

A Java thread can wait for the termination of another Java thread t by calling t.join(). This call blocks the calling thread until the execution of t is termi-nated. There are three variants of this method:

void join(): the calling thread is blocked until the target thread is termi-nated;

void join (long timeout): the calling thread is blocked until the target thread is terminated or the given time interval timeout has passed; the time interval is given in milliseconds;

void join (long timeout, int nanos): the behavior is similar to void join (long timeout); the additional parameter allows a more exact specification of the time interval using an additional specification in nanoseconds.
The calling thread will not be blocked if the target thread has not yet been started.

The method **boolean isAlive()** of the Thread class gives information about the execution status of a thread: The method returns true if the target thread has been started but has not yet been ter-minated; otherwise, false is returned. The join() and isAlive() methods have no effect on the calling thread.
A name can be assigned to a specific thread and can later be retrieved by using the methods:

void setName (String name);
String getName();

An assigned name can later be used to identify the thread. A name can also be assigned at thread creation by using the constructor Thread (String name). The Thread class defines static methods which affect the calling thread or provide information about program execution:

static Thread currentThread();
static void sleep (long milliseconds);
static void yield();
static int enumerate (Thread[] th_array);
static int activeCount();

Since these methods are static, they can be called without using a target Thread object. The call of currentThread() returns a reference to the Thread object of the calling thread. This reference can later be used to call non-static methods of the Thread object. The method sleep() blocks the execution of the calling thread until the specified time interval has passed; at this time, the thread again becomes ready for execution and can be assigned to an execution core or processor.The method yield() is a directive to the Java Virtual Machine (JVM) to assign another thread with the same priority to the processor. If such a thread exists, then the scheduler of the JVM can bring this thread to execution. The use of yield() is useful for JVM implementations without a time-sliced scheduling, if threads per-form long-running computations which do not block. The method enumerate() yields a list of all active threads of the program. The return value specifies the num-ber of Thread objects collected in the parameter array th array. The method activeCount() returns the number of active threads in the program. The method can be used to determine the required size of the parameter array before calling enumerate().

Synchronization of Java Threads

The threads of a Java program access a shared address space. Suitable synchro-nization mechanisms have to be applied to avoid race conditions when a variable is accessed by several threads concurrently. Java provides synchronized blocks and methods to guarantee mutual exclusion for threads accessing shared data. A synchronized block or method avoids a concurrent execution of the block or method by two or more threads. A data structure can be protected by putting all accesses to it into synchronized blocks or methods, thus ensuring mutual exclu-sion.

A synchronized increment operation of a counter can be realized by the fol-lowing method incr():

```
public class Counter
 {
private int value = 0;
public synchronized int incr()
 {
 value = value + 1;
```

return value;
}
}

Java implements the synchronization by assigning to each Java object an implicit mutex variable. This is achieved by providing the general class Object with an implicit mutex variable. Since each class is directly or indirectly derived from the class Object, each class inherits this implicit mutex variable, and every object instantiated from any class implicitly possesses its own mutex variable. The activa-tion of a synchronized method of an object Ob by a thread t has the following effects:

When starting the synchronized method, t implicitly tries to lock the mutex variable of Ob. If the mutex variable is already locked by another thread s, thread t is blocked. The blocked thread becomes ready for execution again when the mutex variable is released by the locking thread s. The called synchronized method will only be executed after successfully locking the mutex variable of Ob.When t leaves the synchronized method called, it implicitly releases the mutex variable of Ob so that it can be locked by another thread.

A synchronized access to an object can be realized by declaring all methods access-ing the object as synchronized. The object should only be accessed with these methods to guarantee mutual exclusion.

Deadlocks

The use of fully synchronized classes avoids the occurrence of race conditions, but may lead to deadlocks when threads are synchronized with different objects. This is illustrated in Fig. 3 for a class Account which provides a method swapBalance() to swap account balances. A deadlock can occur when swapBalance() is executed by two threads *A* and *B* concurrently: For two account objects a and b, if *A* calls a.swapBalance(b) and *B* calls b.swap Balance(a) and *A* and *B* are executed on different processors or cores, a dead-lock occurs with the following execution order:

```
pulic class Account {
  private long balance;
  synchronized long getBalance() {return balance;}
  synchronized void setBalance(long v) {
    balance = v;
  }
  synchronized void swapBalance(Account other) {
    long t = getBalance();
    long v = other.getBalance();
    setBalance(v);
    other.setBalance(t);
  }
}
```

Fig 3: Example of a deadlock

time T_1: thread *A* calls a.swapBalance(b) and locks the mutex variable of object a;
time T_2: thread *A* calls getBalance() for object a and executes this function;
time T_2: thread *B* calls b.swapBalance(a) and locks the mutex variable of object b;
time T_3: thread *A* calls b.getBalance() and blocks because the mutex vari-able of b has previously been locked by thread *B*;
time T_3: thread *B* calls getBalance() for object b and executes this function;
time T_4: thread *B* calls a.getBalance() and blocks because the mutex vari-able of a has previously been locked by thread *A*.

The execution order is illustrated in Fig. 4 After time T_4, both threads are blocked: Thread *A* is blocked, since it could not acquire the mutex variable of object b. This mutex variable is owned by thread *B* and only *B* can free it. Thread *B* is blocked, since it could not acquire the mutex variable of object a. This mutex vari-able is owned by thread *A*, and only *A* can free it. Thus, both threads are blocked and none of them can proceed; a deadlock has occurred.

Deadlocks typically occur if different threads try to lock the mutex variables of the same objects in different orders. For the example in Fig. 4 , thread A tries to lock first a and then b, whereas thread B tries to lock first b and then a.

Time	operation Thread *A*	Operation Thread *B*	Owner mutex a	Owner mutex b
T_1	a.swapBalance(b)		*A*	–
T_2	t = getBalance()	b.swapBalance(a)	*A*	*B*
T_3	Blocked with respect tob	t = getBalance()	*A*	*B*
$^T4$		Blocked with respect toa	*A*	*B*

Fig 4 Execution order to cause a deadlock situation for the class in Fig. 3

In this situation, a deadlock can be avoided by a backoff strategy or by using the same locking order for each thread, . A unique ordering of objects can be obtained by using the Java method System.identityHashCode() which refers to the default implementation Object.hashCode(). But any other unique object ordering can also be used. Thus, we can give an alternative formulation of swapBalance() which avoids deadlocks, see Fig.6. The new formulation also contains an alias check to ensure that the operation is only exe-cuted if different objects are used. The method swapBalance() is not declared synchronized any more.

```
public void swapBalance(Account other) {
  if (other == this) return;
  else if (System.identityHashCode(this) <
      System.identityHashCode(other))
    this.doSwap(other);
  else other.doSwap(this);
}
protected synchronized void doSwap(Account other) {
  long t = getBalance();
  long v = other.getBalance();
  setBalance(v);
  other.setBalance(t);
}
```

Fig 5 Deadlock-free implementation of swapBalance() from Fig. 3

For the synchronization of Java methods, several issues should be considered to make the resulting programs efficient and safe:
Synchronization is expensive. Therefore, synchronized methods should only be used for methods that can be called concurrently by several threads and that may manipulate common object data.

If an application ensures that a method is always executed by a single thread at each point in time, then a synchronization can be avoided to increase efficiency.

Synchronization should be restricted to critical regions to reduce the time interval of locking. For larger methods, the use of synchronized blocks instead of synchronized methods should be considered.

Several Java classes are internally synchronized; examples are Hashtable, Vector, and StringBuffer. No additional synchronization is required for objects of these classes.

If an object requires synchronization, the object data should be put into private or protected instance fields to inhibit non-synchronized accesses from out-side. All object methods accessing the instance fields should be declared as synchronized.

OpenMP

OpenMP is a portable standard for the programming of shared memory systems. The OpenMP API (application program interface) provides a collection of com-piler directives, library routines, and environmental variables. The compiler direc-tives can be used to extend the sequential languages Fortran, C, and C++ with single program multiple data (SPMD) constructs, tasking constructs, work-sharing constructs, and synchronization constructs. The use of shared and private data is supported. The library routines and the environmental variable control the runtime system.

The OpenMP standard was designed in 1997 and is owned and maintained by the OpenMP Architecture Review Board (ARB). Since then many vendors have included the OpenMP standard in their compilers. Currently most compilers support Version 2.5 from May 2005. The most recent update is Version 3.0 from May 2008. Information about OpenMP and the standard definition can be found at the following web site: http://www.openmp.org.

The programming model of OpenMP is based on cooperating **threads** running simultaneously on multiple processors or cores. Threads are created and destroyed in a **fork–join** pattern. The execution of an OpenMP program begins with a sin-gle thread, the initial thread, which executes the program sequentially until a first parallel construct is encountered. At the parallel construct the initial thread cre-ates a team of threads consisting of a certain number of new threads and the initial thread itself. The initial thread becomes the master thread of the team. This fork operation is performed implicitly. The program code inside the parallel construct is called a **parallel region** and is executed in parallel by all threads of the team. The parallel execution mode can be an SPMD style; but an assignment of different tasks to different threads is also possible. OpenMP provides directives for different execution modes. At the end of a parallel region there is an implicit barrier synchronization, and only the master thread continues its exe-cution after this region (implicit join operation). Parallel regions can be nested and each thread encountering a parallel construct creates a team of threads.

The memory model of OpenMP distinguishes between shared memory and pri-vate memory. All OpenMP threads of a program have access to the same shared memory. To avoid conflicts, race conditions, or deadlocks, synchronization mech-anisms have to be employed, for which the OpenMP standard provides appropri- ate library routines. In addition to shared variables, the threads can also use pri-vate variables in the *threadprivate* memory, which cannot be accessed by other threads.
An OpenMP program needs to include the header file <omp.h>. The compila-tion with appropriate options translates the OpenMP source code into multithreaded code. This is supported by several compilers. The Version 4.2 of GCC and newer versions support OpenMP; the option -fopenmp has to be used. Intel's C++ com-piler Version 8 and newer versions also support the OpenMP standard and provide additional Intel-specific directives. A compiler supporting OpenMP defines the vari-able OPENMP if the OpenMP option is activated.

Compiler Directives

In OpenMP, parallelism is controlled by compiler directives. For C and C++, OpenMP directives are specified with the #pragma mechanism of the C and C++ standards. The general form of an OpenMP directive is:

#pragma omp directive [clauses [ ] ...]

written in a single line. The clauses are optional and are different for different directives. Clauses are used to influence the behavior of a directive. In C and C++, the directives are case sensitive and apply only to the next code line or to the block of code (written within brackets { and }) immediately following the directive.

Parallel Region

The most important directive is the parallel construct with syntax

#pragma omp parallel [clause [clause] ...]
{ // structured block ... *}*

The parallel construct is used to specify a program part that should be executed in parallel. Such a program part is called a *parallel region*. A team of threads is created to execute the parallel region in parallel. Each thread of the team is assigned a unique thread number, starting from zero for the master thread up to the number of threads minus one. The parallel construct ensures the creation of the team but does not distribute the work of the parallel region among the threads of the team. If there is no further explicit distribution of work (which can be done by other directives), all threads of the team execute the same code on possibly different data in an SPMD mode. One usual way to execute on different data is to employ the thread number also called *thread id*. The user-level library routine.

int omp get thread num()

returns the thread id of the calling thread as integer value. The number of threads remains unchanged during the execution of one parallel region but may be different for another parallel region. The number of threads can be set with the clause

num threads(expression)

The user-level library routine

int omp get num threads()

returns the number of threads in the current team as integer value, which can be used in the code for SPMD computations. At the end of a parallel region there is an implicit barrier synchronization and the master thread is the only thread which continues the execution of the subsequent program code.
The clauses of a parallel directive include clauses which specify whether data will be private for each thread or shared among the threads executing the parallel region. Private variables of the threads of a parallel region are specified by the private clause with syntax

private(list of variables)
where list of variables is an arbitrary list of variables declared before.

Shared variables of the team of threads are specified by the shared clause with the syntax

shared(list of variables)
where list of variables is a list of variables declared before.
The default clause can be used to specify whether variables in a parallel region are *shared* or *private* by default. The clause
default(shared)

causes all variables referenced in the construct to be shared except the private vari-ables which are specified explicitly. The clause
default(none)

requires each variable in the construct to be specified explicitly as *shared* or *private*.

Example The program code in Fig.6 uses a parallel construct for a par-allel SPMD execution on an array x. The input values are read in the function initialize() by the master thread. Within the parallel region the variables x and npoints are specified as *shared* and the variables iam, np, and mypoints are specified as *private*. All threads of the team of threads executing the parallel region store the number of threads in the variable np and their own thread id in the variable iam. The private variable mypoints is set to the number of points assigned to a thread. The function compute subdomain() is executed by each thread of the team using its own private variables iam and mypoints. The actual computations are performed on the shared array x.

```
#include <stdio.h>
#include <omp.h>

int npoints, iam, np, mypoints;
double *x;

int main() {
  scanf("%d", &npoints);
  x = (double *) malloc(npoints * sizeof(double));
  initialize();
  #pragma omp parallel shared(x,npoints) private(iam,np,mypoints)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    mypoints = npoints / np;
    compute_subdomain(x, iam, mypoints);
  }
}
```

Fig. 6: OpenMP program with parallel construct

A nesting of parallel regions by calling a parallel construct within a parallel region is possible. However, the default execution mode assigns only one thread to the team of the inner parallel region. The library function

void omp set nested(int nested)

with a parameter nested 0 can be used to change the default execution mode to more than one thread for the inner region. The actual number of threads assigned to the inner region depends on the specific OpenMP implementation.

Coordination and Synchronization of Threads

A parallel region is executed by multiple threads accessing the same shared data, so that there is need for synchronization in order to protect critical regions or avoid race condition. OpenMP offers several constructs which can be used for synchronization and coordination of threads within a parallel region. The critical construct specifies a **critical region** which can be executed only by a single thread at a time. The syntax is

#pragma omp critical [(name)]
structured block

An optional name can be used to identify a specific critical region. When a thread encounters a critical construct, it waits until no other thread executes a critical region of the same name name and then executes the code of the critical region. Unnamed critical regions are considered to be one critical region with the same unspecified name. The barrier construct with syntax

#pragma omp barrier

can be used to synchronize the threads at a certain point of execution. At such an explicit barrier construct all threads wait until all other threads of the team have reached the barrier and only then they continue the execution of the subsequent program code.

The atomic construct can be used to specify that a single assign-ment statement is an **atomic operation**. The syntax is

#pragma omp atomic
statement
and can contain statements of the form

x binop= E,
x++, ++x, x--, --x,

with an arbitrary variable x, a scalar expression E not containing x, and a binary operator binop ∈ *{*+, -, *, /, &, ^, |, <<, >>*}*. The atomic construct ensures that the storage location x addressed in the statement belonging to the construct is updated atomically, which means that the load and store operations for x are atomic but not the evaluation of the expression E. No interruption is allowed between the load and store operations for variable x. However, the atomic construct does not enforce exclusive access to x with respect to a critical region specified by a critical construct. An advantage of the atomic construct over the critical construct is that parts of an array variable can also be specified as being atomically updated. The use of a critical construct would protect the entire array.

Locking Mechanism

The OpenMP runtime system also provides runtime library functions for a synchro-nization of threads with the **locking mechanism**. The specific locking mechanism of the OpenMP library provides two kinds of lock variables on which the locking runtime routines operate. *Simple locks* of type omp lock t can be locked only once. *Nestable locks* of type omp nest lock t can be locked multiple times by the same thread. OpenMP lock variables should be accessed only by OpenMP locking routines. A lock variable is initialized by one of the following initialization routines:

void omp_init_lock (omp_lock t *lock)

void omp_init_nest_lock (omp_nest_lock t *lock)

for simple and nestable locks, respectively. A lock variable is removed with the routines

void omp_destroy_lock (omp_lock_t *lock)

void omp_destroy_nest_lock (omp_nest_lock_t *lock).

An initialized lock variable can be in the states *locked* or *unlocked*. At the begin-ning, the lock variable is in the state *unlocked*. A lock variable can be used for the synchronization of threads by locking and unlocking. To lock a lock variable the functions

void omp_set_lock (omp_lock_t *lock)

void omp_set_nest_lock (omp_nest_lock_t*lock)

are provided. If the lock variable is available, the thread calling the lock routine locks the variable. Otherwise, the calling thread blocks. A simple lock is available when no other thread has locked the variable before without unlocking it. A nestable lock variable is available when no other thread has locked the variable without unlocking it or when the calling thread has locked the variable, i.e., multiple locks for one nestable variable by the same thread are possible counted by an internal counter. When a thread uses a lock routine to lock a variable successfully, this thread is said to *own* the lock variable. A thread owning a lock variable can unlock this variable with the routines

void omp unset lock (omp lock t *lock)

void omp unset nest lock (omp nest lock t *lock).

For a nestable lock, the routine omp unset nest lock () decrements the inter-nal counter of the lock. If the counter has the value 0 afterwards, the lock variable is in the state *unlocked*. The locking of a lock variable without a possible blocking of the calling thread can be performed by one of the routines

void omp_test_lock (omp_lock_t*lock)
void omp_test_nest_lock (omp_nest_lock_t *lock)

for simple and nestable lock variables, respectively. When the lock is available, the routines lock the variable or increment the internal counter and return a result value
When the lock is not available, the test routine returns 0 and the calling thread is not blocked.

Message-Passing Programming

The message-passing programming model is based on the abstraction of a parallel computer with a distributed address space where each processor has a local memory to which it has exclusive access. There is no global memory. Data exchange must be performed by message-passing: To transfer data from the local memory of one processor *A* to the local memory of another processor *B*, *A* must send a message containing the data to *B*, and *B* must receive the data in a buffer in its local memory. To guarantee portability of programs, no assumptions on the topology of the interconnection network is made. Instead, it is assumed that each processor can send a message to any other processor.

The Message-Passing Interface (MPI) is a standardization of a message-passing library interface specification. MPI defines the syntax and semantics of library routines for standard communication patterns. Language bindings for C, C++, Fortran-77, and Fortran-95 are sup-ported. In the following, we concentrate on the interface for C and describe the most important features.

There are two versions of the MPI standard: MPI-1 defines standard communication operations and is based on a static process model. MPI-2 extends MPI-1 and provides additional support for dynamic process management, one-sided communication, and parallel I/O. MPI is an interface spec-ification for the syntax and semantics of communication operations, but leaves the details of the implementation open. Thus, different MPI libraries can use differ-ent implementations, possibly using specific optimizations for specific hardware platforms. For the programmer, MPI provides a standard interface, thus ensuring the portability of MPI programs. Freely available MPI libraries are MPICH (see www-unix.mcs.anl.gov/mpi/mpich2), LAM/MPI (see www.lam-mpi. org), and OpenMPI (see www.open-mpi.org).

An MPI pro-gram consists of a collection of processes that can exchange messages. For MPI-1, a static process model is used, which means that the number of processes is set when starting the MPI program and cannot be changed during program execution. Thus, MPI-1 does not support dynamic process creation during program execution. Such a feature is added by MPI-2.

Normally, each processor of a parallel system executes one MPI process, and the number of MPI processes started should be adapted to the number of processors that are available. Typically, all MPI

processes execute the same program in an SPMD style. In principle, each process can read and write data from/into files. For a coordinated I/O behavior, it is essential that only one specific process perform the input or output operations. To support portability, MPI programs should be written for an arbitrary number of processes. The actual number of processes used for a specific program execution is set when starting the program.

On many parallel systems, an MPI program can be started from the command line. The following two commands are common or widely used:
mpiexec -n 4 programname programarguments
mpirun -np 4 programname programarguments.

This call starts the MPI program programname with $p = 4$ processes.

Some semantic terms that are used for the description of MPI operations:

Blocking operation: An MPI communication operation is *blocking*, if return of control to the calling process indicates that all resources, such as buffers, spec-ified in the call can be reused, e.g., for other operations. In particular, all state transitions initiated by a blocking operation are completed before control returns to the calling process.

Non-blocking operation: An MPI communication operation is non-blocking, if the corresponding call may return before all effects of the operation are com-pleted and before the resources used by the call can be reused. Thus, a call of a non-blocking operation only starts the operation. The operation itself is com-pleted not before all state transitions caused are completed and the resources specified can be reused.

The terms *blocking* and *non-blocking* describe the behavior of operations from the *local* view of the executing process, without taking the effects on other processes into account. But it is also useful to consider the effect of communication operations from a *global* viewpoint. In this context, it is reasonable to distinguish between *synchronous* and *asynchronous* communications:

Synchronous communication: The communication between a sending process and a receiving process is performed such that the communication operation does not complete before both processes have started their communication operation. This means in particular that the completion of a synchronous send indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation.

Asynchronous communication: Using asynchronous communication, the send-er can execute its communication operation without any coordination with the receiving process.

MPI Point-to-Point Communication

In MPI, all communication operations are executed using a communicator. A communicator represents a communication domain which is essentially a set of processes that exchange messages between each other. The MPI communicator, MPI COMM WORLD, captures all processes executing a parallel program.
The most basic form of data exchange between processes is provided by point-to-point communication. Two processes participate in this communication opera-tion: A sending process executes a send operation and a receiving process exe-cutes a corresponding receive operation. The send operation is *blocking* and has the syntax:

```
int MPI Send(void *smessage,
int count,
MPI Datatype datatype,
int dest,
int tag,
MPI Comm comm).
```

smessage specifies a send buffer which contains the data elements to be sent in successive order;
count is the number of elements to be sent from the send buffer;
datatype is the data type of each entry of the send buffer; all entries have the same data type;
dest specifies the rank of the target process which should receive the data; each process of a communicator has a unique rank; the ranks are numbered from 0 to the number of processes minus one;

tag is a message tag which can be used by the receiver to distinguish different messages from the same sender;
comm specifies the communicator used for the communication.
The size of the message in bytes can be computed by multiplying the number count of entries with the number of bytes used for type datatype. The tag parameter should be an integer value between 0 and 32,767. Larger values can be permitted by specific MPI libraries.

To receive a message, a process executes the following operation:

int MPI Recv(void *rmessage,
int count,
MPI Datatype datatype,
int source,
int tag,
MPI Comm comm,
MPI Status *status).

This operation is also blocking. The parameters have the following meaning:
rmessage specifies the receive buffer in which the message should be stored;
count is the maximum number of elements that should be received;
datatype is the data type of the elements to be received;
source specifies the rank of the sending process which sends the message;
tag is the message tag that the message to be received must have;
comm is the communicator used for the communication;
status specifies a data structure which contains information about a message after the completion of the receive operation.

The predefined MPI data types and the corresponding C data types are shown in Table 1. There is no corresponding C data type for MPI PACKED and MPI BYTE. The type MPI BYTE represents a single byte value. The type MPI PACKED is used by special MPI pack operations.

Table 1 Predefined data types for MPI

MPI Datentyp	C-Datentyp
MPI CHAR	signed char
MPI SHORT	signed short int
MPI INT	signed int
MPI LONG	signed long int
MPI LONG LONG INT	long long int
MPI UNSIGNED CHAR	unsigned char
MPI UNSIGNED SHORT	unsigned short int
MPI UNSIGNED	unsigned int
MPI UNSIGNED LONG	unsigned long int
MPI UNSIGNED LONG LONG	unsigned long long int
MPI FLOAT	Float
MPI DOUBLE	Double
MPI LONG DOUBLE	long double
MPI WCHAR	wide char
MPI PACKED	special data type for packing
MPI BYTE	single byte value

By using source = MPI ANY SOURCE, a process can receive a message from any arbitrary process. Similarly, by using tag = MPI ANY TAG, a process can receive a message with an arbitrary tag. In both cases, the status data structure contains the information, from which process the message received has been sent and which tag has been used by the sender. After completion of MPI Recv(), status contains the following information:

status.MPI SOURCE specifies the rank of the sending process;
status.MPI TAG specifies the tag of the message received;
status.MPI ERROR contains an error code.

The status data structure also contains information about the length of the mes-sage received. This can be obtained by calling the MPI function

int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count ptr),

where status is a pointer to the data structure status returned by MPI_Recv(). The function returns the number of elements received in the variable pointed to by count ptr.

Internally a message transfer in MPI is usually performed in three steps:
The data elements to be sent are copied from the send buffer smessage speci-fied as parameter into a system buffer of the MPI runtime system. The message is assembled by adding a header with information on the sending process, the receiving process, the tag, and the communicator used.
The message is sent via the network from the sending process to the receiving process.
At the receiving side, the data entries of the message are copied from the system buffer into the receive buffer rmessage specified by MPI_Recv().

Both MPI_Send() and MPI_Recv() are *blocking, asynchronous* operations. This means that an MPI_Recv() operation can also be started when the corre-sponding MPI_Send() operation has not yet been started. The process executing the MPI_Recv() operation is blocked until the specified receive buffer contains the data elements sent. Similarly, an MPI_Send() operation can also be started when the corresponding MPI_Recv() operation has not yet been started. The process executing the MPI_Send() operation is blocked until the specified send buffer can be reused. The exact behavior depends on the specific MPI library used.

The following two behaviors can often be observed:

If the message is sent directly from the send buffer specified without using an internal system buffer, then the MPI_Send() operation is blocked until the entire message has been copied into a receive buffer at the receiving side. In particular, this requires that the receiving process has started the corresponding MPI_Recv() operation.

If the message is first copied into an internal system buffer of the runtime system, the sender can continue its operations as soon as the copy operation into the sys-tem buffer is completed. Thus, the corresponding MPI_Recv() operation does not need to be started. This has the advantage that the sender is not blocked for a long period of time. The drawback of this version is that the system buffer needs additional memory space and that the copying into the system buffer requires additional execution time.

Non-blocking Operations and Communication Modes

The use of blocking communication operations can lead to waiting times in which the blocked process does not perform useful work. For example, a process executing a blocking send operation must wait until the send buffer has been copied into a system buffer or even until the message has completely arrived at the receiving process if no system buffers are used. Often, it is desirable to fill the waiting times with useful operations of the waiting process, e.g., by overlapping communica-tions and computations. This can be achieved by using non-blocking communication operations.

A non-blocking send operation initiates the sending of a message and returns control to the sending process as soon as possible. Upon return, the send operation has been started, but the send buffer specified cannot be reused safely, i.e., the trans-fer into an internal system buffer may still be in progress. A separate completion operation is provided to test whether the send

operation has been completed locally. A non-blocking send has the advantage that control is returned as fast as possible to the calling process which can then execute other useful operations. A non-blocking send is performed by calling the following MPI function:

int MPI_Isend (void *buffer,
int count,
MPI_Datatype type,
int dest,
int tag,
MPI_Comm_comm,
MPI_Request *request).

The parameters have the same meaning as for MPI_Send(). There is an additional parameter of type MPI_Request which denotes an opaque object that can be used for the identification of a specific communication operation. This request object is also used by the MPI runtime system to report information on the status of the communication operation.
A non-blocking receive operation initiates the receiving of a message and returns control to the receiving process as soon as possible. Upon return, the receive operation has been started and the runtime system has been informed that the receive buffer specified is ready to receive data. But the return of the call does not indicate that the receive buffer already contains the data, i.e., the message to be received cannot be used yet. A non-blocking receive is provided by MPI using the function

int MPI_Irecv (void *buffer,
int count,
MPI_Datatype type,
int source,
int tag,
MPI_Comm comm,
MPI_Request *request)

where the parameters have the same meaning as for MPI_Recv(). Again, a request object is used for the identification of the operation. Before reusing a send or receive buffer specified in a non-blocking send or receive operation, the calling process must test the completion of the operation. The request objects returned are used for the identification of the communication operations to be tested for completion. The following MPI function can be used to test for the completion of a non-blocking communication operation:

int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status).

The call returns flag = 1 (true), if the communication operation identified by request has been completed. Otherwise, flag = 0 (false) is returned. If request denotes a receive operation and flag = 1 is returned, the parameter status contains information on the message received as described for MPI_Recv(). The parameter status is undefined if the specified receive operation has not yet been completed. If request denotes a send operation, all entries of status except status.MPI_ERROR are undefined. The MPI function

int MPI_Wait(MPI_Request *request, MPI_Status *status)

can be used to wait for the completion of a non-blocking communication opera-tion. When calling this function, the calling process is blocked until the operation identified by request

has been completed. For a non-blocking send operation, the send buffer can be reused after MPI_Wait() returns. Similarly for a non-blocking receive, the receive buffer contains the message after MPI_Wait() returns.

MPI also ensures for non-blocking communication operations that messages are non-overtaking. Blocking and non-blocking operations can be mixed, i.e., data sent by MPI_ Isend() can be received by MPI_Recv() and data sent by MPI_Send() can be received by MPI_Irecv().